Санкт-Петербургский государственный университет

Направление Математическое обеспечение и администрирование информационных систем
Кафедра Системного Программирования

Анкаренко Сергей Александрович

Осуществление взаимодействия между сервером и аппаратными и программными ресурсами клиента

Дипломная работа

Научный руководитель:
д. ф.-м. н, профессор Терехов А. Н.

Рецензент:
руководитель продуктового направления DocsVision Пуцин С. А.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Ankarenko Sergey Aleksandrovich

Implementation of interaction between server and client software and client hardware

Graduation Thesis

Scientific supervisor:
professor Andrey Terehov

Reviewer:
head of product department DocsVision Sergey Putsin

Saint-Petersburg
2017

Оглавление

Введение

В настоящее время, веб-приложения, управляемые из браузера, по уровню производительности находятся на уровне настольных приложений, а в некоторых областях по функциональности и удобству использования опережают их. Однако остаются такие области, где веб-приложение имеет ряд существенных недостатков относительно приложения, работающего на локальном компьютере. Одним из главных недостатков является отсутствие доступа к программным и аппартным средствам на локальном компьютере.

Это обуславливается защитной технологией браузера, которая называется песочницей[1][1]. Данная технология накладывает серьезные ограничения на выполняемый программный код в браузере, в том числе и на доступ к программным и аппаратным средствам на локальном компьютере. На текущий момент существуют технологии, которые позволяют преодолеть данное ограничение. Примером может служить продукт компании Oracle Java applets[2]. Несмотря на возможность работы с любым браузером, данный продукт имеет ряд серьёзных недостатков, главными из которых являются необходимость установки дополнительного программного обеспечения, небезопасность и отсутствие поддержки[2]. Таким образом, в настоящее время существует необходимость в оптимальном решении, удовлетворяющем требованиям безопасности, производительности и расширяемости.

Необходимо также упомянуть, что современные браузеры находятся на стадии развития и предлагают новые возможности для выполняемого кода. Однако эти возможности предоставляются только одним браузером, для другого браузера приходится искать новое решение. Например, разработчики браузера Chrome разработали технологию Native Messaging[3], предоставляющую API для общения браузерных приложений и расширений с приложениями на локальном компьютере. Данная технология способна решить проблему по обеспечению доступа к про-

[1] англ. Sandbox

[2] https://www.theverge.com/2016/1/28/10858250/oracle-java-plugin-deprecation-jdk-9

граммным и аппаратным средствам на локальном компьютере. Однако данная технология не является стандартом, то есть она поддерживается только в браузерах Chrome. Другие популярные браузеры, такие как Opera или Mozilla Firefox, данное решение не поддерживают. Для таких браузеров необходимо искать альтернативный подход. Возможно, другие браузеры поддерживают свою технологию, позволяющую решить проблему по обеспечению доступа к программным и аппартным средствам на локальном компьютере.

Разумеется, можно разработать отдельную реализацию для отдельного браузера. Однако разработка решения под конкретный браузер сильно ограничивает возможности приложения и приводит к резкому повышению трудозатрат на разработку, отладку и выпуск приложения при увелечении поддерживаемых браузеров с собственной реализацией под каждый. Это делает продукт менее конкурентноспособным за счет повышения трудозатрат и увеличения длительности разработки. Поэтому для крупных компаний, которые создают сложные программные продукты, данное решение крайне невыгодно. Одной из таких компаний является Docsvision[3], под руководством которой выполняется данная работа.

Таким образом, целесообразно разработать решение, которое позволит получить ранее сказанное преимущество веб-приложений и при этом устранит наиболее серьезные недостатки, такие как отсутствие доступа к аппаратным и программным средствам на локальном компьютере. Также установлено, что проблему необходимо решить с учетом обеспечения кроссбраузерности и кроссплатформенности, стремясь минимизировать трудозатраты на разработку и обеспечение программного продукта.

[3]http://www.docsvision.com

1. Постановка задачи

Целью данной работы является создание прототипа, позволяющего взаимодействовать с аппаратными и программными средствами на локальном компьютере в контексте задач электронного документооборота.

Для достижения этой цели был сформулирован следующий набор задач.

- Провести обзор существующих технологий и методов, обеспечивающих взаимодействие между веб-приложением и аппаратными и программными ресурсами на компьютере.
- Разработать архитектуру прототипа.
- Реализовать разработанную архитектуру.
- Выполнить тестирование.

2. Обзор

2.1. Обзор альтернативных подходов и технологий

В данной разделе представлены технологии, позволяющие получить доступ к программным и аппаратным средствам на локальном компьютере.

Новый стандарт разметки веб-страниц, HTML5 [4], был завершен в 2014 году. В контексте данной работы актуально нововведение в стандарте, касающееся доступа к интерфейсу USB. Веб-приложение может получить доступ к ограниченному списку HID устройств: таких как игровой контроллер, микрофон или камера. Использовать данную технологию не предоставляется возможным. Во-первых, из-за ограниченного списка устройств, к которым можно получить доступ. Во-вторых, данная технология не предоставляет доступ к программным ресурсам на локальном компьтере.

Новая инициатива разработчиков Google, WebUSB [5], которая позволяет веб-приложению получать доступ к системным интерфейсам на локальном компьютере. Была выпущена в апреле 2016 года и находится на стадии тестирования [6]. К сожалению, данное решение не подходит по нескольким причинам. Во-перых, это тестовый статус проекта. Во-вторых, не является стандартом, то есть не все браузеры могут поддержать данную инициативу. В-третьих, чтобы получить доступ к устройству, оно должно обладать специальным специальным VendorID, прописанным в ROM памяти при изготовлении на заводе, это значит, что пока нет возможности соединения со старыми устройствами [4].

Технология Native Messaging [3], позволяющая веб-приложению по определённому интерфейсу взаимодействовать с прикладным программным приложением, установленным на компьютере. Минусы данной технологии заключаются в том, что данная технология поддерживается только в браузерах Chrome и Opera, разработчики Safari и вовсе не

[4]Hardware manufacturers will have to update the firmware in their USB devices in order to enable WebUSB access to their device via the Platform Capability descriptor. Later a Public Device Registry will be created so that hardware manufacturers can support WebUSB on existing devices.

планируют ее внедрять [5].

Технология Java applets [2], позволяющая выполнять байт код Java на стороне клиента с помощью виртуальной машины JVM. Одним из минусов данного подхода является необходимость пользователя установить виртуальную машину JAVA. Однако самым главным ее недостатком является неактуальность, большинство современных браузеров попросту ее больше не поддерживают[6].

2.2. Инструментарий

В этом разделе описаны программные инструменты, используемые в данной работе.

Для создания веб-приложений используется фреймворк ASP.NET MVC5 [9]. Он реализован на основе шаблона проектирования MVC [7], благодаря которому соблюдается принцип разделения ответственности [8]. В дополнение к MVC5 используется фреймворк ASP.NET Web API, позволяющий создать API для веб-приложения.

Для удобного управления состоянием приложения была выбрана библиотека Redux. Как сказано в документации, она является "предсказуемым контейнером состояний для JavaScript приложений" [9]. В данной работе используется основная функциональность библиотеки, а именно модули redux, redux-thunk, redux-logger.

Библиотека ReactJS [10], написанная на языке программирования JavaScript, позволяет создавать пользовательские графические пользовательские интерфейсы для веб-приложений. В данной работе используются основные модули библиотеки – React и ReactDOM.

Необходимо отметить, в крупных проектах использование JavaScript в чистом виде повышает сложность разработки ПО из-за своей процедурной направленности и динамической типизации, которая создает множество ошибок в коде. Поэтому компания Microsoft разработала

[5]https://lists.w3.org/Archives/Public/public-hb-secure-services/2016Aug/0001.html
[6]https://www.theverge.com/2016/1/28/10858250/oracle-java-plugin-deprecation-jdk-9
[7]Model View Controller
[8]англ. separation of concerns
[9]https://github.com/reactjs/redux

расширение этого языка – объектно-ориентированный TypeScript [8]. Он и будет использоваться в данной работе.

Для того, чтобы развернуть локальный сервер, используется библиотека Websocket-sharp[10] с открытым исходным кодом, написанная на языке программирования C#. Она позволяет запустить сервер, поддерживающий стандартизированный протокол WebSocket [7], который осуществляет полнодуплексную связь (может передавать и принимать одновременно пакеты) поверх TCP-соединения. Он предназначен для обмена сообщениями между браузером и веб-сервером в режиме реального времени.

Чтобы предоставить веб-странице доступ к ресурсам другого домена, используется технология для современных браузеров CORS [11].

Итог главы

В данной главе представлен обзор существующих технологий и методов, которые являются альтернативным подходом к решению поставленной цели. В результате обзора установлено, что ни одна из существующих технологий не достигает этой цели с учетом таких ограничений, как кроссбраузерность и небольшие затраты ресурсов на производство, отладку и сопровождение.

Также в данной главе представлен вводный обзор технологий, которые использованы в данной работе. Стоит отметить, что при выборе технологий основными критериями являлись актуальность и новизна технологий на момент написания дипломной работы.

[10]https://github.com/sta/websocket-sharp

[11]https://ru.wikipedia.org/wiki/Cross-origin$_r esource_s haring$

3. Архитектура

3.1. Основной подход

В обзоре установлено, что для создания кроссбраузерного решения зависимость веб-приложения от браузера должна быть сокращена до минимума. Все используемые возможности браузера должны являтся частью стандарта.

Подход, применяемый в данной работе, заключается в использовании "посредника" между операционной системой и веб-приложением. Посредником выступает локальное приложение, так как оно имеет доступ к программным и аппаратным ресурсам на локальном компьютере. На рисунке 1, приведена UML диаграмма компонент, демонстрирующая предложенный подход.

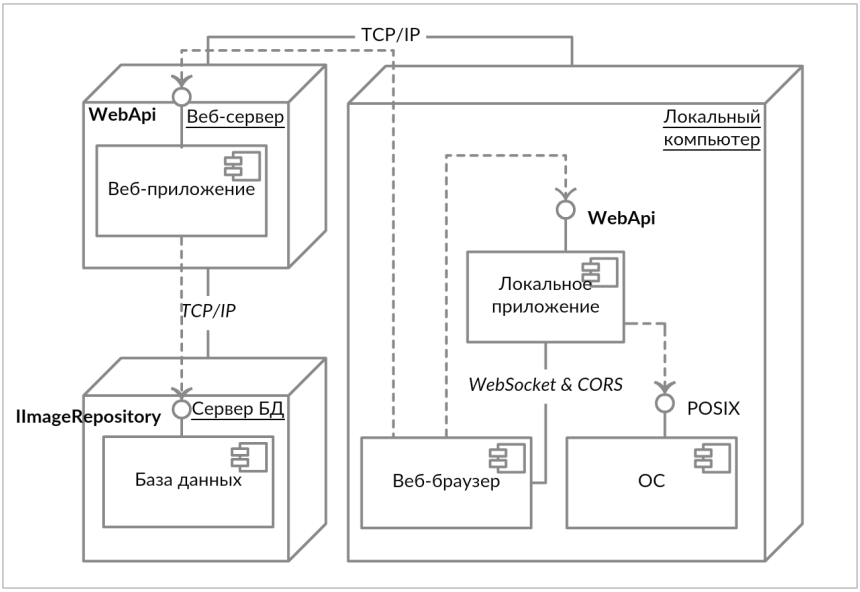

Рис. 1: UML диаграмма компонет.

На диаграмме изображены три взаимодействующих узла: сервер базы данных, веб-приложение, локальный компьютер. Информация между узлами передается по протоколу TCP/IP. Рассмотрим процесс взаимодействия между узлами подробнее.

Пользователь обратился к веб-приложению, ему возвращается клиентское веб-приложение в виде исполняемого программного кода JavaScript

и HTML. Данное клиентское приложение взаимодействует с серверным веб-приложением черз API. Локальное приложение имеет неограниченный доступ к операционной системе. Для его взаимодействия с клиентским веб-приложением разворачивается локальный сервер, поддерживающий протокол WebSocket. Это позволяет клиентскому приложению отправить запрос к локальному приложению и получить данные на основе программных или аппаратных средств локального компьютера. Таким образом, решается проблема по обеспечению доступа к программным и аппаратным ресурсам на локальном приложении.

Сервер базы данных обеспечивает хранение данных веб-приолжения. Доступ к базе данных из веб-приложения обеспечивается через интерфейс IImageRepository, который является "оберткой" над функциями класса, являющегося частью фреймворка с названием Entity Framework 6[12].

Стоит упомянуть, взаимодействие с локальным приложением разрешено только домену веб-приложения, что запрещает злоумышленный доступ к локальному приложению. Это реализовано с помощью технологии CORS.

3.2. Интерфейс веб-приложения

Интерфейс веб-приложения представляет собой взаимодействие двух компонент – серверной и клиентской. Серверная компонента расположена на веб-сервере. А клиентская компонента выполняется в браузере клиента.

На рисунке 2 представлена схема взаимодействия клиентского и серверного веб-приложения.

Основной задачей серверного веб-приложения является предоставление пользователю клиентского веб-приложения и обработка запросов клиентского веб-приложения по работе с базой данных. Разберем компоненты подробнее. На рисунке 3 представлена UML диаграмма классов, демонстрирующая архитектуру серверной части.

[12]https://docs.microsoft.com/en-us/ef

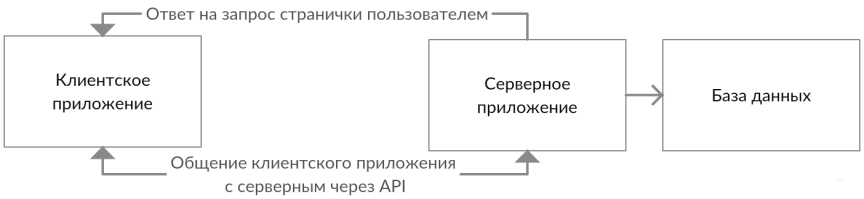

Рис. 2: Взаимодействие клиентской и серверной частей веб-приолжения.

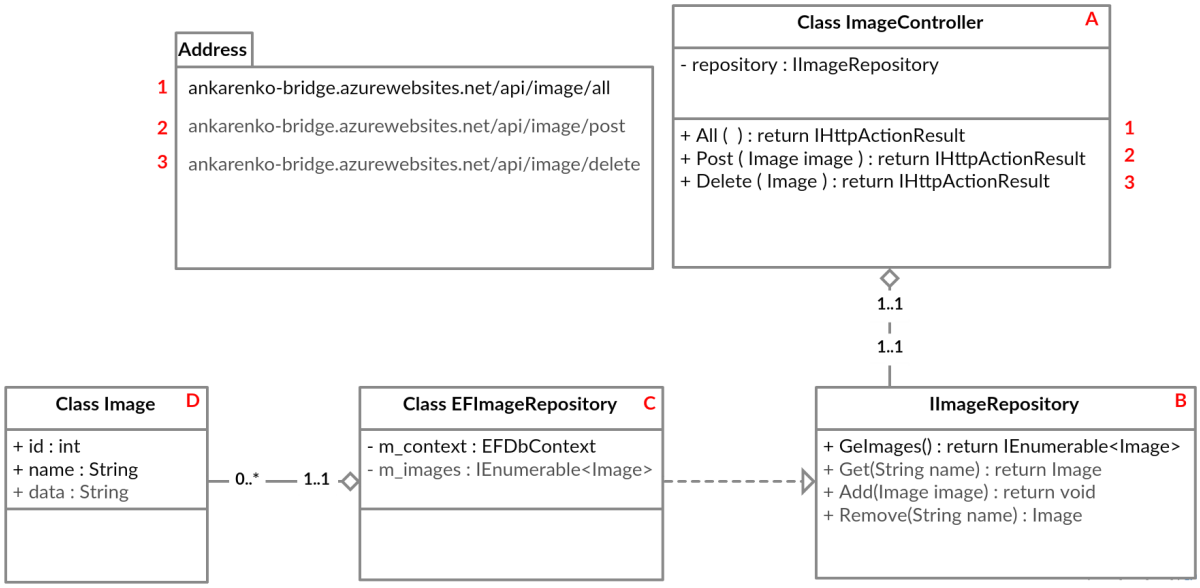

Рис. 3: Backend архитектура

Как было сказано в обзоре, одним из главных компонентов ASP.NET MVC 5 является контроллер, который обозначен буквой A, он реализует интерфейс, продемонстрированный в таблице 1.[13] Данный контроллер

Адрес	Метод	Параметры	Описание
/all	GET	нет	Получить все изображения в базе данных в формате JSON
/post	POST	объект типа Image	Сохранить в базе данных изображение
/remove	DELETE	объект типа Image	Удалить из базы данных изображение

Таблица 1: API веб-приложения

имеет доступ к сервису IImageRepository, который осуществляет взаимодействие с помощью технологии ORM с базой данных.

[13]адрес указан относительно адреса ankarenko-bridge.azurewebsites.net/api/image/

Компонент, обозначаемый буквой D, хранит объект типа EFDBContext для доступа к базе данных. Чтобы программно работать с удаленной БД, используется объектно-ориентированная технология доступа к данным Entity Framework. Она позволяет сопоставить объекты языка программирования C# с объектами реляционной базы данных. Например, объекты класса, обозначенного на рисунке буквой D сопоставляются записям одной из таблиц в БД. Подобное сопоставление делает работу с базой данных удобной в использовании.

Рассмотрим клиентское веб-приложение часть приложения. На рисунке 4 показано взаимодействие Redux и ReactJS посредством использования паттерна MV(Model-View).

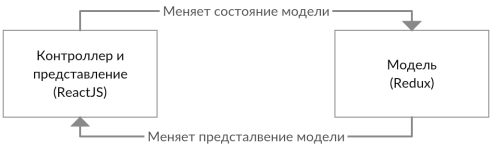

Рис. 4: Взаимодействие Redux и ReactJS.

Модель реализована на основе технологии Redux. Как было сказано в обзоре, данная технология является контейнером состояний приложения. Данные состоянии хранятся в JavaScript объекте. Ниже, на рисунке 5, представлена модель состояния.

Элемент с номером 1 хранит данные о изображениях, информацию о доступных принтерах и сканерах. Необходимо отметить, поле isRequesting. Это позволяет сообщить пользоваетлю о том, что данные находятся в процессе загрузки. Поле под номером 3 необходимо при сканировании данных на локальном компьютере. Поле с номером 4 описывает, выбранное пользователем пункт меню.

Информация из модели передается компонентам ReactJS. Необходимо отметить, что компоненты ReactJS позволяют хранить состояние и реагировать на его изменение. Однако в данной работе они используются лишь в качестве представлений. Управление состоянием приложения делегировано классам из библиотеки Redux.

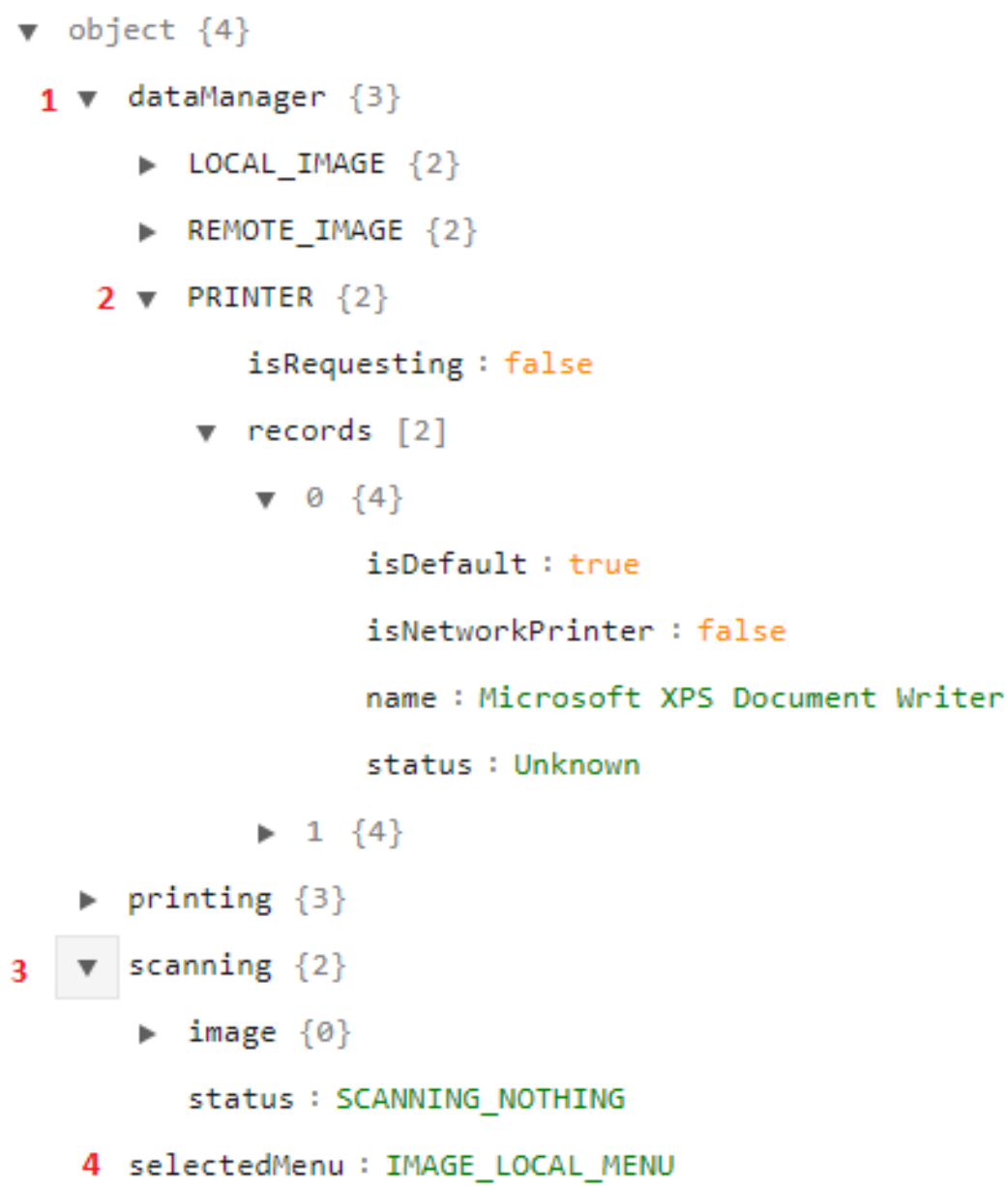

Рис. 5: Redux модель состояния.

3.3. Локальное приложение

Основной задачей при построении архитектуры локального приложения являлась расширяемость. Это обусловлено тем, что в процессе сопровождения будут возникать задачи по добавлению новой функциональности. Например, получение информации о статусе обновления базы данных антивируса. Поэтому была выбрана библиотека WebSocket-sharp, которая позволяет гибкое и расширяемое приложение. На рисун-

ке 6 представлена архитектура локального приложения в виде UML диаграммы.

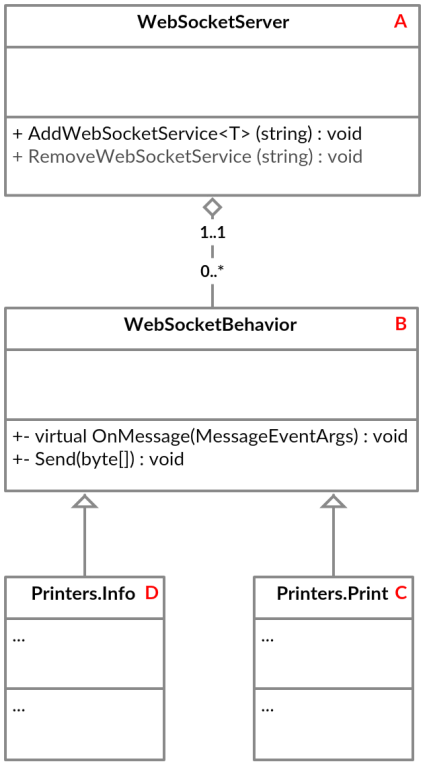

Рис. 6: Архитектура локального приложения.

Компонент A позволяет развернуть сервер, поддерживающий протокол Websocket. Он хранит список сервисов, которые обрабатывают запросы клиента. На рисунке данный компонент изображен под буквой B. Классы, обозначенные буквами C и D, реализованы в данной работе. Они составляют интерфейс для управления программными и аппаратными средствами на локальном компьютере. На диаграмме показано, что эти сервисы не зависят друг от друга. Таким образом, имеется возможность добавить новые изолированные и независимые сервисы.

Ниже в таблице 2 представлен интерфейс, который представляет локальное приложение.

Итог главы

В данной главе подробно описаны основные компоненты архитектуры протитипа, показан процесс их взаимодействия. Далее в работе

Адрес	Параметры	Описание
/Images/Update	нет	Получить все изображения на локальном компьютере в формате JSON
/Images/Edit	индификатор изображения	Отформатировать изображние на локальном компьютере
/Images/Delete	индификатор изображения	Удалить изображение на локальном компьютере
/Printers/Print	объект типа Image	Распечатать изображение на локальном компьютере
/Printers/Info	нет	Получить информацию о доступных принтерах на локлаьном компьютере
/Scanners/Scan	нет	Отсканировать на локлаьном компьютере и получить изображение

Таблица 2: API локального приложения

представлена архитектура вышеописанных компонент с использованием актуальных и передовых на момент написания работы технологий, таких как ReactJS, Redux, ASP.NET MVC5. Необходимо отметить, что учтены основные требования при разработке архитектуры – расширяемость, кроссбраузерность, кроссплатформенность.

4. Реализация

4.1. Взаимодействие ReactJS и Redux

В виду того, что ReactJS и Redux являются независимыми технологиями, потребовалось разработать способ их взаимодействия.

Отметим, что библиотека Redux позволяет подписаться на изменение модели, за счет чего при изменении модели вызывается набор функций, которые были подписаны на изменения. В данном случае ими является функции отрисовки визуального компонента ReactJS. Необходимо отметить, что при этом возникает проблема, которая заключается в том, что таких компонент много и в зависимости от того, какие параметры модели изменились, соответствующие компоненты должны отрисоваться. Например, при выборе в меню подменю настройки, поле в модели "выбранное меню" изменится с "главное меню" на "настройки". Подобное изменение модели влечет перерисовку компонентов: компонент "главное меню" изчезает, а компонент "настройки" появляется на экране.

Для реализации вышеописанного подхода реализован отдельный класс Manager, который подписывается на любое изменение состояния. Когда состояние изменяется, класс Manager анализирует его и отрисовывает нужные компоненты, передавая им параметры, соответствующие текущему состоянию модели.

4.2. Dependency Injection

Одна из наиболее полезных особенностей MVC паттерна заключается в том, что он позволяет разделить приложение на независимые друг от друга компоненты. В идеальной ситуации компоненты ничего не должны знать друг о друге. Взаимодействие происходит исключительно через интерфейсы.

К сожалению, язык C# не предоставляет иного способа создавать интерфейсы, кроме как создавать конкретный объект класса с помощью ключевого слова new. Это может серьезно нарушить несвязность

компонент. То есть при изменении или замене одной из компонент, остальный также будут нуждаться в редактировании.

Например, в реализации веб-приложения используется контроллер ImageController, который работает с базой данных. Он хранит интерфейс для доступа к ней. Допустим мы определили интерфейс таким образом, как это на рисунке 7.

```
1  class ImageController
2  {
3    private IImageRepository mRepository;
4
5    public ImageController()
6    {
7      IImageRepository mRepository = new EFDBImageRepository();
8    }
9  }
```

Рис. 7: До внедрения внешней зависимости.

Тогда при изменении базы данных, то есть модели, необходимо будет редактировать и контроллер. Если переписать пример в виде, представленном на рисунке 8.

```
1  class ImageController
2  {
3    private IImageRepository mRepository;
4
5    public ImageController(IImageRepository newRepository)
6    {
7      mRepository = newRepository;
8    }
9  }
```

Рис. 8: После внедрения внешней зависимости.

Кажется, что проблема решена. Нигде в коде не упоминается конкретная реализация интерфейса. Данный прием называется Dependency Injection[14]. Однако функция, которая инициализирует объект ImageController, должна создать интерфейс IImageRepository с помощью ключевого слова new. То есть вышеописанная проблема не решена, но ”перенесена” на уровень повыше.

[14]https://en.wikipedia.org/wiki/Dependency_injection

Чтобы решить данную проблему, в работе используется Dependency Injection Container, который называется Ninject [15]. Данный контейнер действует, как посредник между классом, который создает зависимость (ImageController) и классом, который ее разрешает, то есть класс, в котором непосредственно используется ключевое слово new. Теперь я могу в одном месте задать все конкретные реализации интерфейса IImageReposito а когда мне понадобится данный интерфейс, я обращуюсь к Ninject контейнеру.

Стоит отметить, что внедрение подобного контейнера очень упрощает процесс тестирования. Например, при необходимости тестирования контроллера можно указать в одном месте Ninject контейнеру разрешать зависимости, используя конкретную реализацию, предназнаечнную для тестирования.

Итог главы

В данной главе представлены основные сложности при реализации прототипа и пути их разрешения. Решена проблема по реализации взаимодействия технологий ReactJS и Redux. С помощью внедрения внешних зависимостей (Dependency injection) компоненты серверного приложения стали более независимы, что благотворно сказывается на дальнейшем тестированиии веб-приложения.

[15] http://www.ninject.org

5. Тестирование

5.1. ”Ручное” тестирование

Чтобы продемонстрировать возможность доступа из веб-приложения к программным и аппаратным средствам на локальном компьютере, в прототипе реализован следующий набор функций.

- Сканирование файлов на локальном компьютере и передача их веб-приложению.
- Печать файла на локальном компьютере из веб-приложения.
- Изменение файлов на локальном компьютере и отправка отредактированных файлов в веб-приложение.

Ниже, на рисунке 9, приведен пример веб-интерфейса.

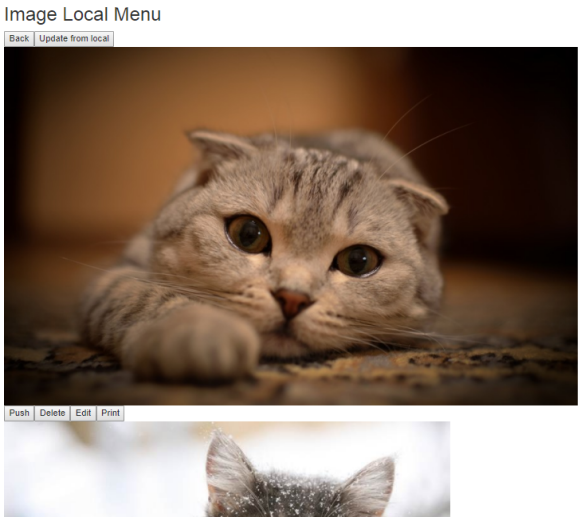

Рис. 9: Интерфейс веб-приложения.

При непосредственном ”ручном” тестировании проблем не было выявлено. Стоит отметить, задачей прототипа, прежде всего, является обеспечения доступа к программным и аппаратным средствам на локальном компьютере. По этой причине он не поддерживает регистрацию или какое-либо многопользовательское взаимодействие, что сильно

упрощает задачу как "ручного", так и unit-тестирования. "Ручное" тестирование свелось к прямой проверке вышеописанной функциональности.

5.2. Unit-тестирование

5.2.1. Серверное приложение

Для тестирования веб-приложения используются встроенные unit-тесты, стандартные для Visual Studio. Для тестирования создается отдельный проект.

По причине того, что веб-приложение создано с использованием технологии ASP.NET MVC5, главный акцент при тестировании сделан на проверку компонентов-контроллеров. А именно, проверку того, что при обращении по определенному адресу вызывается правильный метод в контроллере и что он производит нужные и правильные действия.

При тестировании компонент важно, чтобы они были изолированы и не зависили от окружения. Представим метод контроллера, который принимает интерфейс для доступа базе данных и считает сумму хранящихся в ней элементов. Нашей задачей стоит тестирование отдельного метода.

В данном случае функция имеет внешнюю зависимость (интерфейс доступа к базе данных). Допустим, при разработке было решено использовать другую базу данных или изменить логику методов интерфейса. В таком случае результаты тестирования могут оказаться неудовлетворительными, несмотря на то, что тестируемая функция не менялась и целью тестирования является именно она. Более того, если таких зависимостей будет несколько и результатом тестов будет провал, то окажется совершенно неясным, проблема возникла в тестируемом методе или в его зависимостях.

Чтобы решить данную проблему, в работе используются Moq[9] объекты. Их целью является имитация интерфейса, от которого зависит функция.

В данной работе реализован один класс-контроллер в серверном

приложении. Необходимо было протестировать его методы. Данные методы имеют одну внешнюю зависимость – интерфейс для доступа к базе данных.

Ниже, на рисунке 10, приведен пример одного из тестов.

```
1  [TestMethod]
2  public void can_get_all_images()
3  {
4      //инициализация
5      Mock<IImageRepository> mock = new Mock<IImageRepository>();
6      ImageController controller = new ImageController(mock.Object);
7      IEnumerable<Image> images = new List<Image>
8      {
9          new Image() { id = 0, data = "data_0", name="name_0" },
10         new Image() { id = 1, data = "data_1", name="name_1" },
11         new Image() { id = 2, data = "data_2", name="name_2" }
12     };
13     mock.Setup(m => m.m_images).Returns(images);
14
15     //применение
16     OkNegotiatedContentResult<ResponseModel<IEnumerable<Image>>> contResult =
17         (OkNegotiatedContentResult<ResponseModel<IEnumerable<Image>>>)
18         controller.All();
19     IEnumerable<Image> imagesResult = contResult.Content.data;
20
21     //проверка
22     Assert.AreEqual(imagesResult, images);
23 }
```

Рис. 10: Тестирование возможности контроллера возвращать данные.

Данным тестом проверяется, что при вызове метода All в контроллере пользователю возвращается содержимое базы данных. В строке 5 создается Mock объект, задачей которого является имитация базы данных, при запросе имеющихся данных данный объект вернет фиксированную коллекцию. Такое поведение определяется в строке 13. Далее происходят типичные для Unit тестирования этапы – вызов тестироемого метода или функции и проверка результатов.

Подобные Unit-тесты были написаны для каждого метода в контроллере.

5.2.2. Клиентское приложение

Как было упомянуто в главе Архитектура, клиентское приложение реализовано с использованием таких технологий, как ReactJS и Redux.

Так как компоненты библиотеки React отвечают лишь за представление данных и не реализуют никакую логику, unit-тестирование для них не представляется возможным. Поэтому основной акцент был сделан на тестирование модели, которая реализована с помощью библиотеки Redux. Одна из причин, по которой эта технология была выбрана, заключается в простоте тестирования.

Как было сказано в обзоре, Redux является контейнером состояний. Состояние приложения меняется с помощью ограниченного набора функций, которые называются редьсерами[16]. Они принимают текущее состояние и событие в качестве параметра и возвращают новое состояние. Данные функции имеют ряд ограничений. Данные функции должны быть "чистыми", то есть они не зависят от окружения, не меняют входные параметры. Из данных ограничений следует, что они хорошо тестируемы, так как проблем с зависимостями, которые встречаются при тестировании серверного приложения, не возникнет. Также стоит заметить, что для тестирования работы всего приложения достаточно протестировать функции-редьюсеры, так как только они способны менять состояние приложения. Логика этих тестирующих их функций проста и однообразна, на рисунке 11 представлен пример тестирующего метода.

```
function IMAGE_RECORD_REMOTE_REQUEST() {
    let BEFORE = {
        isRequesting:false,
        records:[1, 2, 3, 4]
    };
    let AFTER = {
        isRequesting:true,
        records:[1, 2, 3, 4]
    };
    let ACTION = {
        type:K.REQUEST
    };
    deepFreeze(BEFORE);
    expect(REDUCER.processSubdata(BEFORE, ACTION)).toEqual(AFTER);
}
```

Рис. 11: Пример тестирования метода-редьюсера.

На этапе инициализации определяются состояние до события, со-

[16]англ. reducers

бытие и состояние после события. Далее происходит вызов функции и проверка результатов. На рисунке этот этап выполняется в строке 14.

Заметим, что в данном тесте использовалась функция deepFreeze [17]. Если к переменной применена данная функция, то при попытке изменить переменную в консоль будет выведено сообщение об ошибке. Данная функция необходима, чтобы удостовериться в том, что функция-редьюсер не изменяет переменные-параметры.

Подобные тесты были написаны для всех функций-редьюсеров.

Итог главы

В данной главе представлено, как проводилось unit-тестирование. Оказалось, что разыне компоненты тестируются по-разному. Более того, unit-тестирование также зависит от используемой технологии. Хочется сказать, что клиентское и серверное приложения хорошо "покрыты" тестами, что позволяет вносить изменения или добавлять новую функциональность без риска, что ранее работающие функции окажутся неработоспособными. Также в работе было проведено "ручное" тестирование прототипа, которое сводилось к прямой проверке умеющейся функциональности. Как было сказано, основной акцент при "ручном" тестировании делался на проверку возможности прототипа обеспечевать доступ к программным и аппатным средствам на локальном компьютере через веб-приложение.

[17] https://github.com/substack/deep-freeze

Заключение

В ходе работы были достигнуты следующие результаты.

- Проведен обзор технологий HTML5, WebUSB, JavaApplets, Native Messaging. В результате обзора установлено, что ни одно из существующих решений не способно решить поставленную цель с учетом требования кроссбраузерности.

- Разработана архитектура прототипа. В ходе решения данной задачи были подробно разработаны основные компоненты архитектуры. Создан процесс их взаимодействия. Архитектура разрабатывалась с использованием самых актуальных и передовых на момент написания дипломной работы технологий – ReactJS, Redux, ASP.NET MVC5. Также разработка велась с учетом ограничений кроссплатформенности и расширяемости.

- Реализована архитектура прототипа с помощью таких технологий, как ASP.NET, Redux, ReactJS. При реализации разработанной архитектуры возникло немало проблем. Основными из которых были обеспечение взаимодействия технологий ReactJS и Redux, внедрение внешних зависимостей с целью разбиения компонент на независимые, хорошо тестируемые части.

- Проведено тестирование. В результате чего клиентское и серверное приложения полностью "покрыты" тестами, это позволяет вносить изменения или добавлять новую функциональность без риска, что ранее работающие функции окажутся неработоспособны. Успешно проведено "ручное" тестирование.

Список литературы

[1] URL: https://en.wikipedia.org/wiki/Sandbox_(computer_security) (online; accessed: 24.05.2017).

[2] URL: http://www.oracle.com/technetwork/java/applets-137637.html (online; accessed: 24.05.2017).

[3] URL: https://developer.chrome.com/extensions/nativeMessaging (online; accessed: 12.29.2016).

[4] URL: https://en.wikipedia.org/wiki/HTML5 (online; accessed: 12.29.2016).

[5] URL: https://wicg.github.io/webusb/ (online; accessed: 12.29.2016).

[6] URL: https://www.chromestatus.com/feature/5651917954875392 (online; accessed: 12.29.2016).

[7] Alexey Melnikov Ian Fette. The WebSocket Protocol. — 2011. — URL: https://tools.ietf.org/html/rfc6455 (online; accessed: 12.20.2016).

[8] Fenton Steve. Pro TypeScript. — 2014.

[9] Freeman Adam. Pro ASP.NET MVC 5. — 2013.

[10] Stefanov Stoyan. React: Up Running: Building Web Applications. — 2017.